A Monadic Analysis of Information-Flow Security with Mutable State
Enforcing Secrecy with Types

Aleksey Kliger
joint work with
Karl Crary and Frank Pfenning

October 21, 2005 / Student Seminar Series
Helen shops for a house
LHH.com
Running untrusted code

Helen wants
A guarantee that Luke cannot learn income

Luke wants
Reasonable burden of proof
Running untrusted code

Language-based information flow security.
Type system tracks flow of information.
Well typed programs do not leak secrets.

Helen wants
A guarantee that Luke cannot learn income

Luke wants
Reasonable burden of proof
Information flow

Luke thinks
Someone wants
a house in
Oakland
Luke thinks
Someone wants a house in Oakland
Information flow

Luke thinks

Someone wants a house in Oakland
Information flow

Luke thinks
Someone wants a cheap house in Oakland
Abstracting away

- Untrusted program
- Private data
- Public data
- Must access both
Abstracting away

Situation

- Untrusted program
- Private data
- Public data
- Must access both
Abstracting away
Abstracting away

Guarantee
No information flow of high-security data to low-security observer.
Abstracting away

Guarantee
The high-security data does not interfere with the low-security behavior of the program.
Non-interference
Non-interference
Outline

1 Introduction
   Motivation
   Abstracting Away

2 Types of Information Flow
   Direct Information Flow
   Indirect: Control Flow

3 Monads
   Suspensions and Effects
   Monadic Security
   Informativeness

4 Related and Future Work
Direct information flow
First try at LHH.com

Example

```javascript
fun processForm () {
    houses := fetchHouses (zipcodeField);
    priceRange := calcPriceRange (incomeField);
    showJustAffordable (houses, priceRange);
}
```
Direct information flow
First try at LHH.com

Example

```java
fun processForm () {
  houses :=
      fetchHouses (zipcodeField);  // In: ⌁, ⌁, Out: ⌁, ⌁. ✔
  priceRange :=
      calcPriceRange (incomeField);
  showJustAffordable (houses,
       priceRange);
}
```
Example

```kotlin
fun processForm () {
    houses := fetchHouses (zipcodeField);
    priceRange := calcPriceRange (incomeField);  In: 🔒, Out: 🔒. ✔
    showJustAffordable (houses, priceRange);
}
```
Direct information flow
First try at LHH.com

Example

```haskell
fun processForm () {
    houses := fetchHouses (zipcodeField);
    priceRange := calcPriceRange (incomeField);
    showJustAffordable (houses, priceRange);
    In:  Out:  .
}
```
Example

```kotlin
fun processForm () {
    houses := fetchHouses (zipcodeField);
    priceRange := calcPriceRange (incomeField);
    showJustAffordable (houses,
                         priceRange);
    In: , Out: . ?
}
```
Direct information flow
First try at LHH.com

Example

```kotlin
fun processForm() {
    houses = fetchHouses(zipcodeField);
    priceRange = calcPriceRange(incomeField);
    showJustAffordable(houses, priceRange);
}
```

To Combine Inputs: Maximum

\[ \max(\text{houses}, \text{priceRange}) = \text{priceRange} \]

In: ✔ Out: ✔️
Example (What if **houses** is secret)

```java
fun processForm () {
    houses :=
        fetchHouses (zipcodeField);
    priceRange :=
        calcPriceRange (incomeField);
    showJustAffordable (houses,
                       priceRange);
}
```
Example (What if houses is secret)

```javascript
fun processForm () {
  houses :=
    fetchHouses (zipcodeField);  \ In: \(\), \ Out: \(\), ?
  priceRange :=
    calcPriceRange (incomeField);
  showJustAffordable (houses, priceRange);
}
```
Direct information flow
Another try at LHH.com

Example (What if houses is secret)

```kotlin
fun processForm() {
    houses :=
        fetchHouses (zipcodeField);  // In: , Out: ✔
    priceRange :=
        calcPriceRange (incomeField);
    showJustAffordable (houses, priceRange);
}
```

To Combine Outputs: Minimum

```
min(houses, priceRange) =
```

Types of Information Flow
Direct information flow
Another try at LHH.com

Example (What if \textit{houses} is secret)

```java
fun processForm () {
    houses := fetchHouses (zipcodeField);
    priceRange := calcPriceRange (incomeField);  \textit{In: \textbullet, Out: \textbullet. ✔}
    showJustAffordable (houses, priceRange);
}
```
Example (What if houses is secret)

```java
fun processForm () {
    houses :=
        fetchHouses (zipcodeField);
    priceRange :=
        calcPriceRange (incomeField);
    showJustAffordable (houses,
        priceRange);
    In: ✔, ✔ Out: ✔.
}
```
Indirect information flow

Example (LHH.com “I’m Feeling Lucky”)

1. Fetch a random house in the zipcode
2. If not in the price range, go back to (1)
3. Otherwise show the house
Indirect information flow

Example (Simplified)

```plaintext
randHouse := fetchRandom (zipcodeField);
if (not (isInPriceRange (randHouse, priceRange)))
    then
        randHouse := fetchRandom (zipcodeField);
showHouse (randHouse)
```

Interaction

⇒

"fetchRandom 15213"

⇐

"fetchRandom 15213"
Indirect information flow

Example (Simplified)

```plaintext
randHouse := fetchRandom (zipcodeField);
if (not (isInPriceRange
    (randHouse,
    priceRange)))
    then
        randHouse := fetchRandom (zipCodeField);
showHouse (randHouse)
```

Interaction

⇒ “fetchRandom 15213”

⇐

Types of Information Flow

Indirect: Control Flow
Indirect information flow

Example (Simplified)

```plaintext
randHouse := fetchRandom (zipcodeField);
if (not (isInPriceRange (randHouse, priceRange)))
  then
    randHouse := fetchRandom (zipCodeField);
showHouse (randHouse)
```

Interaction

⇒ “fetchRandom 15213”

⇐
**Example (Simplified)**

```plaintext
randHouse := fetchRandom (zipcodeField);
if (not (isInPriceRange (randHouse, priceRange)))
  then
    randHouse := fetchRandom (zipCodeField);
showHouse (randHouse)
```

**Interaction**

⇒ “fetchRandom 15213”

←

⇒ “fetchRandom 15213”

Luke: $$$
Indirect information flow

Example (Simplified)

```
randHouse := fetchRandom (zipcodeField);
if (not (isInPriceRange
    (randHouse, priceRange)))
    then
        randHouse := fetchRandom
            (zipcodeField);
    showHouse (randHouse)
```

Interaction

⇒ “fetchRandom 15213”

←
Indirect information flow

Example (Simplified)

```plaintext
randHouse := fetchRandom (zipcodeField);
if (not (isInPriceRange (randHouse, priceRange)))
  then
    randHouse := fetchRandom (zipcodeField);
showHouse (randHouse)
```

Interaction

⇒ “fetchRandom 15213”

⇐
Indirect information flow

Example (Simplified)

\[
\text{randHouse} := \text{fetchRandom (zipcodeField)};
\]
\[
\text{if } (\text{not } (\text{isInPriceRange (randHouse, priceRange)})]
\]
\[
\text{then }
\]
\[
\text{randHouse} := \text{fetchRandom (zipCodeField)};
\]
\[
\text{showHouse (randHouse)}
\]

Interaction

⇒ “fetchRandom 15213”

⇐

⇒ nothing

Luke: $$$$$$
Indirect information flow, cont’d

Example (Simplified)

```plaintext
if (not (isInPriceRange (randHouse, priceRange)))
then
    randHouse := fetchRandom (zipCodeField);
```

Principle

Security level of a conditional is an *implicit input* to the branches.

Example

*In: 🗺️, 🗺️, 🗺️,🔒 Out: 🗺️, 🗺️ ✗*
Outline

1. Introduction
   - Motivation
   - Abstracting Away

2. Types of Information Flow
   - Direct Information Flow
   - Indirect: Control Flow

3. Monads
   - Suspensions and Effects
   - Monadic Security
   - Informativeness

4. Related and Future Work
Why Monads?

Problems with traditional languages

1. Every expression may have an effect (I/O, memory read/write) and this is not reflected in the types
2. Complicated control flow — indirect information flow
A monadic language for security

Definition (Monad)
A value of type $\circ A$ is a suspended computation, that each time it is forced to execute will
- return a (possibly different) result of type $A$
- potentially produce some effects

Note
And expressions of all other types are pure (do not have effects).
A monadic language for security

Definition (Monad)
A value of type $\circ A$ is a suspended computation, that each time it is forced to execute will

- return a (possibly different) result of type $A$
- potentially produce some effects

Forcing suspended computations
No way to force $\circ A$, to get $A$. Only build up larger composite computations.

Runtime forces $\text{main} : \circ()$. 
Writing programs with monads

Example (increment)

\( \text{inc} : (\text{Ptr} \ \text{Int}, \ \text{Int}) \rightarrow \bigcirc \text{Int} \)

\[ \text{fun inc (ptr, amt) =} \]
\[ \text{do} \]
\[ \ldots \]

Values of \( \bigcirc A \)

\[ \text{do} \]
\[ \text{stmt}_1 \]
\[ \text{stmt}_2 \]
\[ \ldots \]
\[ \text{expr} \]
Writing programs with monads

**Statements**

\[ \text{var} \leftarrow \text{expr} \]

**Expressions**

\[ \text{do stmts} \]
\[ \text{deref expr} \]

**Example (increment)**

\[ \text{fun inc} (\text{ptr}, \text{amt}) = \]
\[ \text{do} \]
\[ \text{oldVal} \leftarrow \text{deref ptr} \]
\[ \ldots \]
Writing programs with monads

Statements
var ← expr
expr

Expressions
pre do stmts
deref expr
expr := expr

Example (increment)

\[
\text{inc : } (\text{Ptr Int, Int}) \\
\quad \rightarrow \bigcirc \text{Int}
\]
fun inc (ptr, amt) =
pre do
oldVal ← deref ptr
ptr := oldVal + amt
...
Writing programs with monads

Statements
var ← expr
expr

Expressions
do stmts
deref expr
expr := expr
pure expr

Example (increment)

\[
inc : (\text{Ptr Int, Int}) 
\rightarrow \bigcirc \text{Int}
\]

fun inc (ptr, amt) =
do
oldVal ← deref ptr
ptr := oldVal + amt
pure oldVal
Writing programs with monads

**Statements**
- `var ← expr`
- `expr`

**Expressions**
- `do stmts`
- `deref expr`
- `expr := expr`
- `pure expr`
- `var`
- `if (expr) then expr else expr`
- `func (args)`

**Example (increment)**

```plaintext
inc : (Ptr Int, Int) → □ Int
fun inc (ptr, amt) =
do
  oldVal ← deref ptr
  ptr := oldVal + amt
pure oldVal
```

Monads
Suspensions and Effects
SSS 23 / 56
Control flow and monads

Example

\( \text{repeatUntil} : \)
\[ (A \rightarrow \text{Bool}, \bigcirc A) \rightarrow \bigcirc A \]

fun repeatUntil
  (test, comp) =
do
  x ← comp
if (test (x))
    then pure x
  else repeatUntil
      (test, comp)
Control flow and monads

Example

\( \text{repeatUntil} : \) 
\( (A \to \text{Bool}, \bigcirc A) \to \bigcirc A \)

\textbf{fun} repeatUntil 
\( \) 
\( \) 
\( (\text{test}, \text{comp}) = \) 
\( \) 
\( \) 
\( \text{do} \)
\( \quad x \leftarrow \text{comp} \)
\( \quad \text{if} (\text{test} (x)) \) 
\( \quad \text{then pure } x \)
\( \quad \text{else} \ \text{repeatUntil} \) 
\( \quad \) 
\( \) 
\( \) 

Example (Sample Output)

State = \{ \text{ptr} \mapsto 0 \} 

repeatUntil (above100, inc (ptr, 1))
Control flow and monads

Example

\[
\text{repeatUntil} : (A \to \text{Bool}, \text{\bigcirc}A) \to \text{\bigcirc}A
\]

fun \text{repeatUntil} (test, comp) =
\[
\text{do}
x \leftarrow \text{comp}
\text{if } (\text{test} (x))
\text{then pure } x
\text{else } \text{repeatUntil} (\text{test}, \text{comp})
\]

Example (Sample Output)

State = \{ \text{ptr} \mapsto 0 \}

\[
\text{do}
x \leftarrow \text{inc} (\text{ptr}, 1)
\text{if } (\text{above100} (x))
\text{then pure } x
\text{else } \text{repeatUntil} (\text{above100}, \text{inc} (\text{ptr}, 1))
\]
Control flow and monads

Example

\[ \text{repeatUntil } : \]
\[ (A \rightarrow \text{Bool}, \bigcirc A) \rightarrow \bigcirc A \]

fun repeatUntil
\[ (\text{test}, \text{comp}) = \]
do
\[ x \leftarrow \text{comp} \]
if (test (x))
then pure x
else repeatUntil
\[ (\text{test}, \text{comp}) \]

Example (Sample Output)

State = \{ ptr \mapsto 0 \}
do
⇒ x \leftarrow inc (ptr, 1)
if (above100 (x))
then pure x
else repeatUntil
\[ (\text{above100}, \]
\[ inc(ptr, 1)) \]
Control flow and monads

Example

\( \text{repeatUntil} : (A \to \text{Bool}, \bigcirc A) \to \bigcirc A \)

fun repeatUntil (test, comp) =
  do
    x ← comp
    if (test (x))
      then pure x
    else repeatUntil (test, comp)

Example (Sample Output)

State = \{ptr ↦ 1\}

do
  x ← inc (ptr, 1)
  ⇒ if (above100 (0))
      then pure 0
      else repeatUntil (above100, inc(ptr, 1))
Control flow and monads

Example

\[
\text{repeatUntil} : \quad (A \rightarrow \text{Bool}, \bigcirc A) \rightarrow \bigcirc A
\]

fun repeatUntil
\[
\text{(test, comp)} =
\]
do
\[
x \leftarrow \text{comp}
\]
if (test (x))
\[
\text{then pure } x
\]
else repeatUntil
\[
\text{(test, comp)}
\]

Example (Sample Output)

State = \{ptr \rightarrow 1\}

if (above100 (0))
\[
\text{then pure } 0
\]
else repeatUntil
\[
\text{(above100, inc (ptr, 1))}
\]
Control flow and monads

Example

\( \text{repeatUntil} : (A \to \text{Bool}, \odot A) \to \odot A \)

fun repeatUntil
\((\text{test}, \text{comp})\) =

\( \text{do} \)
\( x \leftarrow \text{comp} \)
\( \text{if } (\text{test}(x)) \)
\( \text{then pure } x \)
\( \text{else } \text{repeatUntil} \)
\( (\text{test}, \text{comp}) \)

Example (Sample Output)

State = \{\text{ptr} \mapsto 1\}

\( \text{repeatUntil} \)
\( (\text{above100}, \text{inc (ptr, 1)}) \)
Control flow and monads

Example

\[
\begin{aligned}
\text{repeatUntil} &: (A \to \text{Bool}, \bigcirc A) \to \bigcirc A \\
\text{fun} \ \text{repeatUntil} &: (\text{test}, \text{comp}) = \\
& \text{do} \\
& \hspace{1em} x \leftarrow \text{comp} \\
& \hspace{1em} \text{if} (\text{test} (x)) \\
& \hspace{2em} \text{then} \ \text{pure} x \\
& \hspace{1em} \text{else} \ \text{repeatUntil} \\
& \hspace{2em} (\text{test}, \text{comp})
\end{aligned}
\]

Example (Sample Output)

\[
\text{State} = \{ \text{ptr} \mapsto 1 \}
\]

\[
\begin{aligned}
\text{repeatUntil} &: (\text{above100}, \text{inc} (\text{ptr}, 1))
\end{aligned}
\]
Monadic version of LHH.com

Original version

```plaintext
fun processForm () {
    houses := fetchHouses (zipcodeField);
    priceRange := calcPriceRange (incomeField);
    showJustAffordable (houses, priceRange);
}
```
Monadic version of LHH.com

Monadic Version

fun processForm () =
do
  houses := fetchHouses (zipcodeField)
  priceRange := calcPriceRange (incomeField)
  showJustAffordable (houses, priceRange)
Monadic version of LHH.com

Types

\[ \text{fetchHouses : } \text{Ptr } \text{ZipCode} \rightarrow \bigcirc \text{ List House} \]

Monadic Version

\[
\text{fun } \text{processForm}() = \\
\text{do} \\
\quad \text{houses} := \text{fetchHouses (zipcodeField)} \\
\quad \text{priceRange} := \text{calcPriceRange (incomeField)} \\
\quad \text{showJustAffordable (houses, priceRange)}
\]
Monadic version of LHH.com

Types

Monadic Version

fun processForm () =
do
    h ← fetchHouses (zipcodeField)
    houses := h
    priceRange := calcPriceRange (incomeField)
    showJustAffordable (houses, priceRange)
Monadic version of LHH.com

Types

calcPriceRange : \text{Int} \rightarrow \text{Range}

Monadic Version

\textbf{fun} \hspace{1em} \text{processForm} () =
\begin{verbatim}
do
h ← fetchHouses (zipcodeField)
houses := h
priceRange := calcPriceRange (incomeField)
showJustAffordable (houses, priceRange)
\end{verbatim}
Monadic version of LHH.com

Types

Monadic Version

fun processForm () =
do
    h ← fetchHouses (zipcodeField)
    houses := h
    i ← deref incomeField
    priceRange := calcPriceRange (i)
    showJustAffordable (houses, priceRange)
Monadic version of LHH.com

Types

\[ \text{showJustAffordable} : (\text{Ptr List House, Ptr Range}) \rightarrow \bigcirc() \]

Monadic Version

fun processForm () =
do
  h ← fetchHouses (zipcodeField)
  houses := h
  i ← deref incomeField
  priceRange := calcPriceRange (i)
  showJustAffordable (houses, priceRange)
Idiomatic monadic version of LHH.com

Types

\(\text{showJustAffordable} : (\text{List House, Range}) \rightarrow \bigcirc()\)

Simplified

fun processForm () =
  do
    h ← fetchHouses (zipcodeField)
    i ← deref incomeField
    showJustAffordable (h, calcPriceRange (i))
Idiomatic monadic version of LHH.com

Simplified

```haskell
fun processForm () =
do’h ← fetchHouses (zipcodeField)
do’i ← deref incomeField
showJustAffordable (h, calcPriceRange (i))
```
So what about security?

Idea

Track security levels of input and output in the monad: ☐️🔒
LHH.com with security monads

Types

Code unchanged

fun processForm () =
do
h ← fetchHouses (zipcodeField)
do
i ← deref incomeField
showJustAffordable (h, calcPriceRange (i))
LHH.com with security monads

Types

\[ \text{fetchHouses} : \textbf{Ptr ZipCode} \rightarrow \bigcirc, \textbf{List House} \]

Code unchanged

\[
\textbf{fun} \quad \text{processForm} () = \\
\textbf{do} \\
\quad h \leftarrow \text{fetchHouses} (\text{zipcodeField}) \\
\textbf{do} \\
\quad i \leftarrow \text{deref incomeField} \\
\quad \text{showJustAffordable} (h, \text{calcPriceRange} (i))
\]
LHH.com with security monads

Types

deref incomeField : • Int

Code unchanged

fun processForm () =
do
  h ← fetchHouses (zipcodeField)
do
  i ← deref incomeField
  showJustAffordable (h, calcPriceRange (i))
LHH.com with security monads

Types

\[ \text{showJustAffordable} : (\text{List} \ \text{House}, \text{Range}) \rightarrow \bigcirc, \bullet, \text{\textbullet}() \]

Code unchanged

```haskell
fun processForm () =
do
  h ← fetchHouses (zipcodeField)
do
  i ← deref incomeField
showJustAffordable (h, calcPriceRange (i))
```
Composing suspensions

Portion of `processForm`

```
do
  i ← `deref` `incomeField` : Int
  `showJustAffordable`
      (h, `calcPriceRange` (i)) : ()
```
Composing suspensions

Portion of processForm

do
  i ← deref incomeField : ⃝, Int
  showJustAffordable
    (h, calcPriceRange (i)) : ⃝, ()
  : ⃝, ()
Composing suspensions, cont’d

processForm

do
  h ← fetchHouses
  (zipcodeField) : ⚠️, List House
  do ... : ⚠️, ()
Composing suspensions, cont’d

processForm

do
    h ← fetchHouses
        (zipcodeField) :  List House
    do ... : ()
  : ()
What are the types telling us?

Suppose `showJustAffordable` had type

\[(\text{List } \text{House, Range}) \rightarrow \bigcirc \cdot \checkmark \text{Int}\]

Consider

\[\text{do}
\begin{align*}
\text{nHouses} &\leftarrow \text{do} \\
&\quad i \leftarrow \text{deref incomeField} \\
&\quad \text{showJustAffordable (\ldots)} \\
&\quad : \bigcirc \checkmark, \checkmark \text{Int} \\
&\quad \text{sendL (nHouses)} : \bigcirc \cdot, \checkmark ()
\end{align*}\]
Definition
A type $A$ is **not informative** at low-security if no computation with only low input could make use of it.

Theorem
*Non-interference is preserved if for any $A$ that is not informative at low-security, we promote $\bigcirc\lock,\lock, A$ to $\bigcirc\bullet,\lock, A$.*
processForm is well-typed

Promote the inner do-block

fun processForm () =
do
  h ← fetchHouses (zipcodeField) : ⃝, List House
do
  i ← deref incomeField
  showJustAffordable (...)
  : ⃝, ()
  : ⃝, ()

Monads
Informativeness
SSS 41 / 56
Other non-informative types?

Not informative for low security

- $A \rightarrow B$ is not informative, whenever $B$ isn’t
- $\text{Ptr}_A$ is not informative
- $\bigcirc_A$ is not informative, whenever $A$ isn’t
Secure computation continuation passing

Example

\[
\text{do}
\]

\[
\text{cont } \leftarrow \text{highComp}
\]

\[
\text{lowComp}
\]

\[
\text{cont}
\]
Secure computation continuation passing

Example

```haskell
do
  cont ← highComp
  lowComp
  cont
```
Is it still secure?
Yes
1. Introduction
   Motivation
   Abstracting Away

2. Types of Information Flow
   Direct Information Flow
   Indirect: Control Flow

3. Monads
   Suspensions and Effects
   Monadic Security
   Informativeness

4. Related and Future Work
Related work
Monads in language design

Foundations

• Semantics [Moggi 1989]

Have been used for:

• I/O in Haskell [Peyton-Jones, et al. 1993]
• Parsing Combinators [Wadler 1992]
• Composable Transactional Memory [Peyton-Jones, et al. 2005]
• Probabilistic Computation [Park et al. 2005]
• and much more...
Related work
Language-based security

Non-interference:
• [Volpano, et al. 1996]
• SLam [Heintze, et al. 1998], DCC [Abadi et al. 1999]
• CoreML$^2$ [Pottier, et al. 2003]
• and many others ... [Sabelfeld, et al. 2003]

Extensions:
• Concurrency [Honda, et al. 2002]
• Timing channels [Agat 2000]
• Robust declassification [Zdancewic, et al. 2001], [Myers, et al. 2004]
• and many others...
Our contributions

- Monads for tracking information flow
- Informativeness
Monads for tracking information flow

Isolate security concerns: simplify reasoning
Only monads and channels are tagged

Example

\[ \text{fetchHouses} : \text{Ptr} \to \text{ZipCode} \rightarrow \bigcirc \to \text{List} \to \text{House} \]
\[ \text{calcPriceRange} : \text{Int} \rightarrow \text{Range} \]
Informativeness

Allow high-security computations to pass temporary results through low-security computations

\[ \circ\left(\text{\textbullet}, \text{\textbullet}\right) \circ\left(\text{\textbullet}, \text{\textbullet}\right)(\cdot) \rightarrow \circ\left(\text{\textbullet}, \text{\textbullet}\right)(\cdot) \]

\[ \circ\left(\text{\textbullet}, \text{\textbullet}\right)(\cdot) \rightarrow \circ\left(\text{\textbullet}, \text{\textbullet}\right)(\cdot) \]

Related and Future Work
Future work

**Push out**  Concurrency (e.g. transactional memory), robust declassification

**Push down**  Security-Typed Assembly Language
Questions?
The three monad laws
Program equivalences

Within each security level, we obey the monad laws
(Upcalls are an additional relationship between monad families)

1. \texttt{do} \ x \leftarrow \texttt{pure} \ expr
   \hspace{1cm} func (x)

2. \texttt{do} \ x \leftarrow \texttt{comp}
   \hspace{1cm} comp \ pure \ x

3. \texttt{do} \ y \leftarrow \texttt{do}
   \hspace{1cm} x \leftarrow \texttt{comp1}
   \hspace{1cm} comp2 (x)
   \hspace{1cm} \texttt{comp3} (y)
## Differences in the paper

### Additions in the paper
- Full lattice of security levels
- A proof of non-interference
- Discussion of allocation
- Encoding of a prior work

### Stylistic differences
- No • for “no interesting reads/writes,” use \( \bot, \top \)
- “A informative only above \( \bullet \)” vs. “A not informative for \( \bullet \)”
- Pfenning-style expressions vs. \texttt{do-notation}